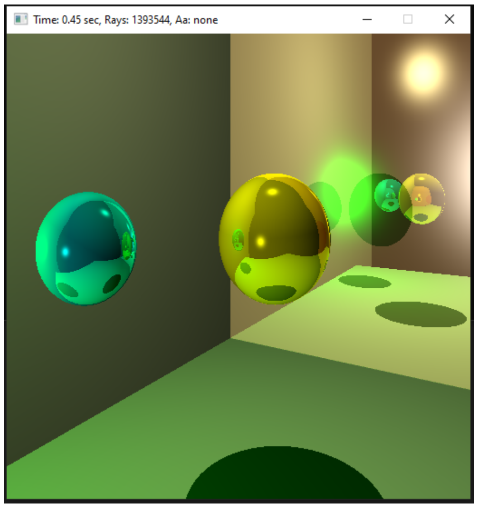
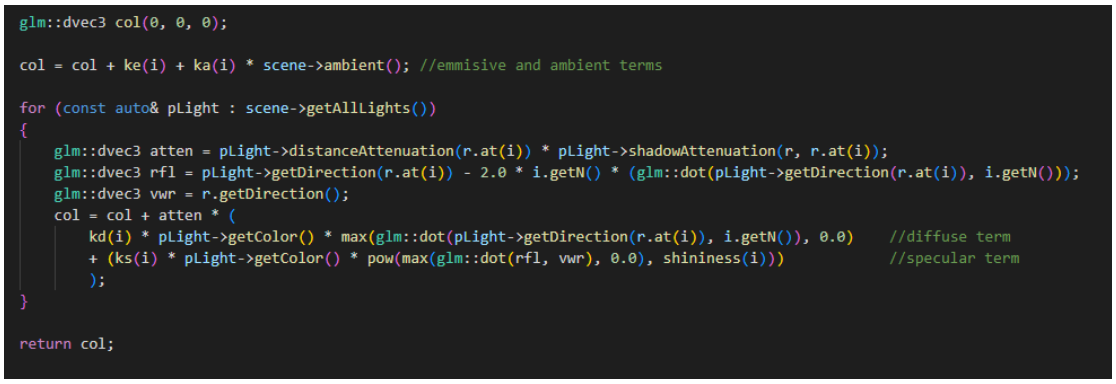
Matthew Jagen
2/7/2023

For this assignment, I implemented Phong shading (with emissive, ambient, diffuse, and specular terms for point and direction lights), recursive reflection rays, recursive refraction rays, triangle-ray intersections with barycentric weights, and normal interpolation using barycentric weights.

Here's an image of the reflection 2 scene that showcases Phong shading and recursive reflection raycasts, and then some code snippets of where it was implemented:



Phong shading and point light attenuation code (directional light attenuation is pretty much exactly the same):



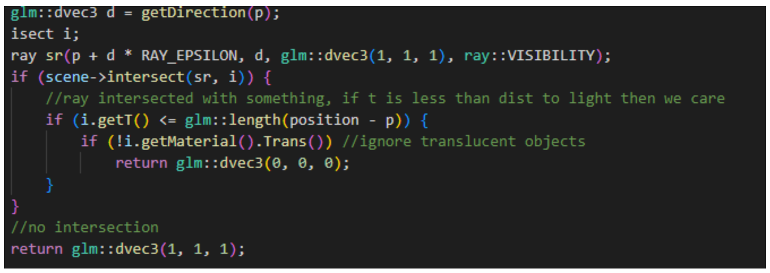
```cpp
glm::dvec3 col(0, 0, 0);

col = col + ke(i) + ka(i) * scene->ambient(); //emmisive and ambient terms

for (const auto& pLight : scene->getAllLights())
{
    glm::dvec3 atten = pLight->distanceAttenuation(r.at(i)) * pLight->shadowAttenuation(r, r.at(i));
    glm::dvec3 rfl = pLight->getDirection(r.at(i)) - 2.0 * i.getN() * (glm::dot(pLight->getDirection(r.at(i)), i.getN()));
    glm::dvec3 vwr = r.getDirection();
    col = col + atten * (
        kd(i) * pLight->getColor() * max(glm::dot(pLight->getDirection(r.at(i)), i.getN()), 0.0)    //diffuse term
        + (ks(i) * pLight->getColor() * pow(max(glm::dot(rfl, vwr), 0.0), shininess(i)))           //specular term
        );
}

return col;
```

shadow attenuation:



```cpp
glm::dvec3 d = getDirection(p);
isect i;
ray sr(p + d * RAY_EPSILON, d, glm::dvec3(1, 1, 1), ray::VISIBILITY);
if (scene->intersect(sr, i)) {
    //ray intersected with something, if t is less than dist to light then we care
    if (i.getT() <= glm::length(position - p)) {
        if (!i.getMaterial().Trans()) //ignore translucent objects
            return glm::dvec3(0, 0, 0);
    }
}
//no intersection
return glm::dvec3(1, 1, 1);
```

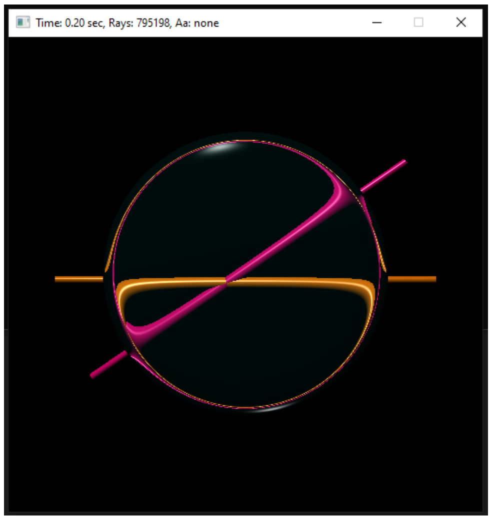
Distance attenuation:

```
double d = glm::length(position - P);
double atten = 1.0 / (constantTerm + linearTerm * d + quadraticTerm * pow(d, 2.0));
```

Reflection code:

```
const Material& m = i.getMaterial();
colorC = m.shade(scene.get(), r, i);
//recursion
if (depth > 0) {//glm::length(thresh))) {
    //reflection
    if (m.Refl()) {
        glm::dvec3 rfldir = r.getDirection() - 2.0 * i.getN() * (glm::dot(r.getDirection(), i.getN()));
        ray rfl(r.at(i) + i.getN() * RAY_EPSILON, glm::normalize(rfldir), glm::dvec3(1, 1, 1), ray::VISIBILITY);
        colorC += m.kr(i) * traceRay(rfl, thresh, depth - 1, t);
    }
```

Here’s an example of my reflection code in action:

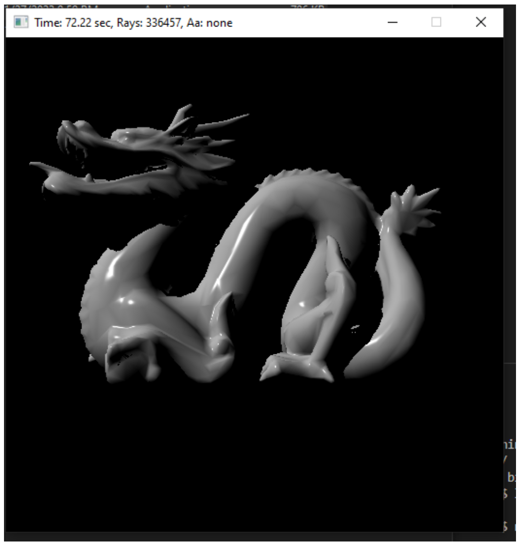


And the corresponding code:

```
//refraction
if (m.Trans()) {
    double n_i = 1.0;
    double n_t = 1.0;
    glm::dvec3 n = i.getN();
    glm::dvec3 e = -r.getDirection();
    bool entering = glm::dot(r.getDirection(), i.getN()) < 0;
    if (entering) {
        n_t = m.index(i);
    }
    else { //exiting object
        n_i = m.index(i);
        n = -n;
    }
    double n_r = n_i / n_t;
    double t_sqrt = 1.0 - pow(n_r, 2) * (1.0 - pow(glm::dot(n, e), 2));
    if ((glm::length(m.kt(i)) > 0) && (t_sqrt > 0)) {
        glm::dvec3 trndir = n * (n_r * glm::dot(n, e) - sqrt(t_sqrt)) - n_r * e;
        ray trn(r.at(i) - n * RAY_EPSILON, glm::normalize(trndir), glm::dvec3(1, 1, 1), ray::VISIBILITY);
        colorC += m.kt(i) * traceRay(trn, thresh, depth - 1, t);
    }
}
```

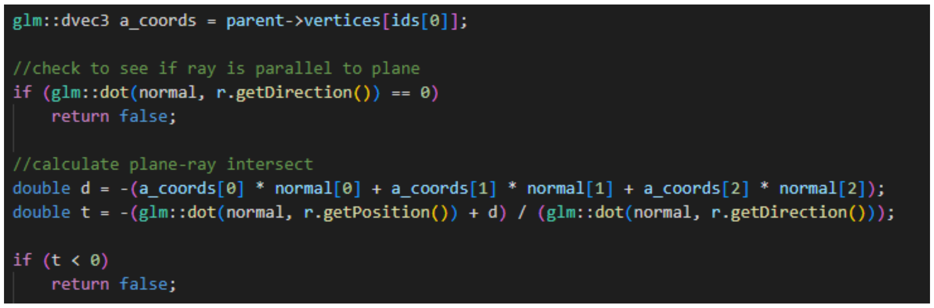
The refraction code also follows the pseudocode closely, but handles the fact that each plane only has one normal pointing outwards, so in the case that we are exiting the object it is reversed so we can use the same calculation.

And lastly, here's dragon 2 to show the trimesh functionality and normal interpolation:



```
Time: 72.22 sec, Rays: 336457, Aa: none
```

First, in TrimeshFace::intersectLocal(), we calculate the t value for which the ray intersects with the trimesh's plane:



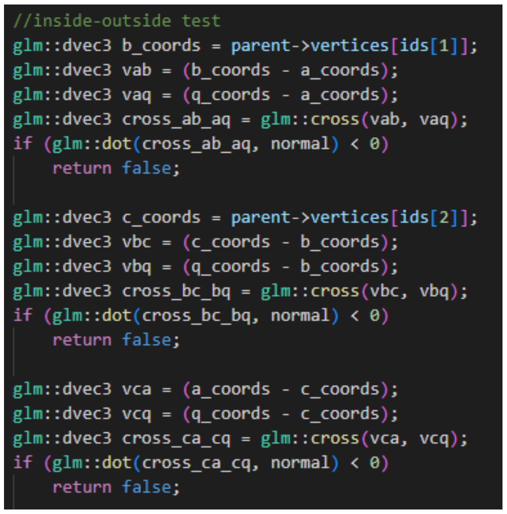
```
glm::dvec3 a_coords = parent->vertices[ids[0]];

//check to see if ray is parallel to plane
if (glm::dot(normal, r.getDirection()) == 0)
    return false;

//calculate plane-ray intersect
double d = -(a_coords[0] * normal[0] + a_coords[1] * normal[1] + a_coords[2] * normal[2]);
double t = -(glm::dot(normal, r.getPosition()) + d) / (glm::dot(normal, r.getDirection()));

if (t < 0)
    return false;
```

Then, after determining that the ray does intersect with the plane, we do the full inside-outside /left-hand test:

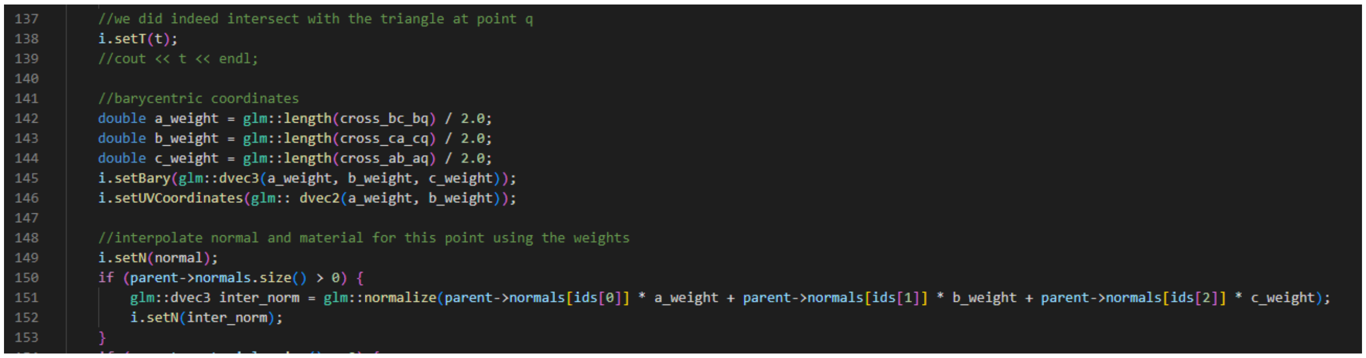


```
//inside-outside test
glm::dvec3 b_coords = parent->vertices[ids[1]];
glm::dvec3 vab = (b_coords - a_coords);
glm::dvec3 vaq = (q_coords - a_coords);
glm::dvec3 cross_ab_aq = glm::cross(vab, vaq);
if (glm::dot(cross_ab_aq, normal) < 0)
    return false;

glm::dvec3 c_coords = parent->vertices[ids[2]];
glm::dvec3 vbc = (c_coords - b_coords);
glm::dvec3 vbq = (q_coords - b_coords);
glm::dvec3 cross_bc_bq = glm::cross(vbc, vbq);
if (glm::dot(cross_bc_bq, normal) < 0)
    return false;

glm::dvec3 vca = (a_coords - c_coords);
glm::dvec3 vcq = (q_coords - c_coords);
glm::dvec3 cross_ca_cq = glm::cross(vca, vcq);
if (glm::dot(cross_ca_cq, normal) < 0)
    return false;
```

If the function has not returned false still at this point, then it means that the point q is inside the trimesh and we did intersect. So, from here we get to work setting the values for the isect parameter.
The barycentric coordinates are calculated and set using the stored results of our inside-outside test earlier, and then if the trimesh has per-vertex normals we go ahead and calculate the normal for point Q given the weights:



```
    //we did indeed intersect with the triangle at point q
    i.setT(t);
    //cout << t << endl;

    //barycentric coordinates
    double a_weight = glm::length(cross_bc_bq) / 2.0;
    double b_weight = glm::length(cross_ca_cq) / 2.0;
    double c_weight = glm::length(cross_ab_aq) / 2.0;
    i.setBary(glm::dvec3(a_weight, b_weight, c_weight));
    i.setUVCoordinates(glm:: dvec2(a_weight, b_weight));

    //interpolate normal and material for this point using the weights
    i.setN(normal);
    if (parent->normals.size() > 0) {
        glm::dvec3 inter_norm = glm::normalize(parent->normals[ids[0]] * a_weight + parent->normals[ids[1]] * b_weight + parent->normals[ids[2]] * c_weight);
        i.setN(inter_norm);
    }
```